# The Level 3 Event Receiver 3.6

**Stephen Tether**

**28 April 2000**

## (I) Interface with the rest of the converter node code

The initial reception of events from Scanner CPUs takes place quasi-independently of the rest of the converter node code, although it runs on the same machines. The header file and function prototypes described below form the sole interface between the two systems (beside the packet buffers, of course). The idea is to keep the interface as "narrow" as possible, to simplify understanding (and debugging). The event receiver library must be linked into the converter node module whose responsibility is reception.

The Run 2 implementation of the receiver package is multi-threaded, using multiple tasks under VxWorks and POSIX threads under UNIX or Linux. The receiver threads share the address space, open files, etc., of the process that uses this interface. In particular, for every buffer managed by this package, N network connections will be established, where N is the number of Scanner CPUs in the event builder; therefore make sure your program will be allowed to obtain all the required resources (e.g., file descriptors).

### ATM under VxWorks

The VxWorks systems in the CDF event builder at B0 use Interphase model 4515 ATM-PMC adapters. VPI and VCI information is sent in structures meant for TCP/IP addresses (struct sockaddr_in ). Special names ending in "-a" have been added to the network host table in each machine. These names are given "IP" addresses which are really VPI numbers. VCI numbers play the same role as TCP port numbers, and are set inside the receiver package; just use 0 for port numbers passed to the package. VxWorks C code to fill in an ATM network address will look like this:

```
#include "netinet/in.h"   /* For struct sockaddr_in */
#include "hostLib.h"      /* For hostGetByName() */
• • •
struct sockaddr_in netaddr;
• • •
netaddr.sin_port = 0;
netaddr.sin_addr.s_addr = hostGetByName("xxxxxx-a");
```

At present ATM is used only for event transmission and not for transmitting startup information; that always goes by the usual LAN.

## Ethernet, FDDI, or other LAN

The Scanner Manager, L3 receiver package, and Scanner CPUs exchange startup information over the LAN (usually Ethernet) via TCP/IP. Again, the receiver package will set port numbers itself, so you should set these to zero. You can also use a LAN for event transmission if you're performing tests on a Linux/GNU workstation (with Scanner Manager and Scanner CPU code compiled with the appropriate options). LAN addresses in `sockaddr_in` structures are obtained in the usual way defined for host. The only potential trouble with using TCP/IP for event transmission is that the boundaries between transmissions on the same connection are not preserved; the receiver only sees a continuous stream of bytes. To get around this the event builder's I/O code creates simulated packets by writing a 32-bit byte count (big-endian) before the contents of the "packet" itself. It does this *only* when compiled for TCP/IP.

## receiver.h

```
#include "receiver.h"
```

Contains prototypes for the functions described below and the definitions of the return code constants.

## set_receiver_timeout()

```
void set_receiver_timeout(int number_of_tries, int try_interval)
```

This function may be called at any time, even before `init_receiver()` (which is recommended). It sets a deadline starting from the arrival of an Event Complete message from the Scanner Manager. All fragments for the given event must arrive within the deadline, which is *number_of_tries * try_interval / 3* milliseconds long. A test for arrival will be repeated every *try_interval / 3* milliseconds before the deadline. If you don't call `set_receiver_timeout()` the receiver package will use a default timeout that will probably be too short.

## init_receiver()

```
int init_receiver(int nbldr, void *buffaddr[], int nbuff,int
partition, int nl3, struct sockaddr_in l3addr[], struct sockaddr_in
smaddr, void* scramnet_addr, int first_connection_id)
```

Sets up the event-receiver package. This function is meant to be called by the converter node after a the partition the receiver will serve has been defined in the event builder. *nbldr* is the number of Builder objects to create. *buffaddr[]* is an array of pointers to packet buffers. *nbuff* is the number of buffers allocated (and the number of elements in *buffaddr[]*). *partition* is the number of the partition for which this receiver will be working (**numbering starts at zero**). *nl3* is the total number of converter nodes in the partition and *l3addr[]* contains their event transmission network addresses.

*smaddr* gives the LAN address for the Scanner Manager. *scramnet_addr* is the base address of the SCRAMNet module to use (as mapped into the address space of the process). first_connection_id should normally be -1. Other values are used during debugging to change the port or VCI numbers used for event transfers.

Unlike previous versions of the receiver package, this version treats each buffer as the container for one network packet, rather than one event. There is no dividing up of big buffers for events into smaller ones for fragments. You can still allocate your buffers in contiguous clusters and pass the individual addresses to `init_receiver()`, but there is no guarantee that all the buffers used for an event will lie in the same cluster. The buffer size should be the maximum packet size `MAX_FRAGMENT_PACKET_SIZE` defined in `evbLimits.h` in the `cdfevb_sm` package. This is 50K for ATM. The test for buffer overlap assumes that all buffers are exactly this size.

The L3 configuration information represented by *partition, nl3,* and *l3addr* isn't used by the receiver itself but is passed to the Scanner Manager via the LAN. Since every converter node must call `init_receiver()` to enable its own event reception, the start-up of a partition with N converter nodes will send this information N times to the Scanner Manager. For each different partition number the SM will take the copy which arrives first and will ignore the others (save for some simple consistency checks).

After setup is complete, LAN network links are closed and messages are passed between the receiver package and the Scanner Manager via a reflective memory network.

Preconditions
(1) The receiver package has not been initialized since the last time (if any) that `kill_receiver()` was called.

(2) The Scanner Manager and Scanner SCPUs must have performed Grand Init for the partition you want to use.

(3) Buffers do not overlap.

(4) The buffers are readable and writable.

(5) Every buffer is exactly `MAX_FRAGMENT_PACKET_SIZE` bytes long.

Postconditions
(1) The next call to `test_next_event()` or `wait_next_event()` will work properly.

(2) The Builders and Boss for this converter node have been constructed and they have been registered with the Scanner Manager.

(3) Connections to the SCPUs have been made over the event transmission network.

(4) The appropriate regions of reflective memory have been found and are initialized.

Returns
`RECV_SUCCESS.`

`RECV_TOOMANY.` Too many buffers were specified.

`RECV_OVERLAP.` Buffers overlapped.

`RECV_BAD_VALUE.` One of the arguments had an absurd value, such as *nbuff* < 0.

`RECV_ALREADY_INITIALIZED`.

`RECV_FAILURE`. Anything not covered by the above. A message will have been printed on the standard error file.

## test_next_event()

```
int test_next_event(recv_event_token* new_event)
```

Checks whether a new event is available. Returns immediately. A `recv_event_token` is a structure defined in `receiver.h` as follows

```
typedef struct {
  int builder_id;
  unsigned long scanner_mask;
  unsigned long tardy_mask;
  unsigned long taginfo[2];
  int firstPacket[MAX_SCANNERS];
  packetInfo packet[MAX_PACKETS_PER_EVENT];
} recv_event_token;
```

```
typedef struct {
  void* address;
  int numBytes;
  int hasHeader;
  int next;
} packetInfo;
```

where `MAX_SCANNERS` and `MAX_PACKETS_PER_EVENT` are defined in `evbLimits.h` of the Scanner Manager package. `MAX_PACKETS_PER_EVENT` is set to allow a maximum event size of about 5 MB. **Never modify the contents of an event token.**

Preconditions
(1) `init_receiver()` has been called, it succeeded, and `kill_receiver()` hasn't been called since then.

Postconditions
(1) If an event is available, RECV_SUCCESS is returned and information about the event is placed in *new_event. That information is

(a) The ID no. of the Builder that received the event,
(b) A bit mask showing which Scanner CPUs contributed to the event,
(c) A bit mask showing which SCPUs should have contributed but didn't,
(d) Two taginfo words, which count the number of commands received from the Scanner Manager (index 0) and the sequence number of the event in the current run (index 1),
(e) Each fragment (contribution from one SCPU) is described as a singly-linked list inside *packet[]*. The entry for the first packet received for fragment *i* is at the index given by *firstPacket[i]*. The *next* field of each entry contains the index for the entry of the next packet received for the fragment. For both *firstPacket[]* and *next* the role of the null pointer is played by -1. If the value of *firstPacket[i]* is -1, then bit *i* of *scanner_mask* is zero, and vice versa. Each entry of packet[] gives the address of the buffer, the number of bytes received from the SCPU for the buffer, and a flag telling whether an SCPU fragment header was detected at the beginning of the packet. For each chain of packets the first and only the first packet ought to contain the SCPU header. The structure of the SCPU header is defined in `fragment.h` (type *fragmentHeader*).

The following example C code finds all the packets for an event and does just a little error checking.

```
recv_event_token *t;
int f, p;
for (f = 0; f < MAX_SCANNERS; f++) {
  p = t->firstPacket[f];
  if (p != -1 && !t->packet[p].hasHeader) {
    printf("Warning. First packet of fragment %d doesn't have "
           "SCPU header.\n", f);
  }
  while(p != -1) {
    processPacket(t->packet[p].address, t->packet[p].numBytes);
    p = t->packet[p].next;
  }
}
```

Bit masks: mask & (1 >>i) != 0 if and only if SCPU i is included.

If the preconditions are not met, or there is no event, a value other than RECV_SUCCESS is returned and the builder ID will be set to -1. Nothing else about the contents of the event token is guaranteed in this case.

Returns

`RECV_SUCCESS.`
`RECV_NO_EVENT.`
`RECV_NOT_INITIALIZED.`

## wait_next_event()

`int wait_next_event(recv_event_token* *new_event*)`

Waits indefinitely for a new event. Otherwise behaves like `test_next_event()`.

## release_event()

`int release_event(recv_event_token* *new_event*)`

Makes an event buffer available again after the event it contains is no longer needed.

Preconditions
(1) `init_receiver()` has been called, it succeeded, and `kill_receiver()` hasn't been called since then.

(2) The event token contents were set by a successful call to `wait_next_event()` or `test_next_event()` and have not yet been used with `release_event()`.

Postconditions
(1) The release request is queued.

Returns
`RECV_SUCCESS.`

`RECV_BAD_BUILDER_ID.` The *builderId* field of the event token is not between 0 and *nbldr*-1, where *nbldr* is the number of Builders argument given to `init_receiver()`.

`RECV_NOT_INITIALIZED.`

## get_receiver_stats()

```
void get_receiver_stats(receiver_stats *)
```

Fills a `receiver_stats` structure with the values of a few counters. The counters are no longer available as global external variables. The definition of `receiver_stats` in receiver.h is

```
typedef struct {
  int l3get;
  int l3free;
  int l3late;
  int l3ioerr;
} receiver_stats;
```

*l3get* is the number of times a buffer has been allocated to hold a packet coming in over the event network. *l3free* is how often such buffers have been released into the pool of empty buffers after the contents have been sent to Level 3 or discarded. *l3late* is the number of fragments that have arrived from SCPUs after the deadline set by `set_receiver_timeout()`. *l3ioerr* counts the number of packet I/O errors for the event network. The counters are not reset by `kill_receiver()` or `init_receiver()`.

## kill_receiver()

`int kill_receiver(void)`

Destroys all the receiver threads and the resources used to manage them. Connections to the event transmission network are closed. Buffers are not altered.

<u>Preconditions</u>

(1) `init_receiver()` has been called, it succeeded, and `kill_receiver()` hasn't been called since then.

<u>Returns</u>

`RECV_SUCCESS.`
`RECV_NOT_INITIALIZED.`

## (II) Outline of the internal workings

This is an outline of the internal structure of the event receiving package for the Level 3 converter nodes. The description is in terms of objects, even though the receiver package is written in a standard procedural language (C). For the purposes of this outline, “object” means a collection of data and related routines that operate on the data. “Data” is interpreted loosely to include threads and network links. The term “network link” may mean a BSD-like socket, a VPI/VCI combination, or any other means of specifying a connection.

Two major types of objects act together to receive events. *Builders* control the flow of information into and out of packet buffers, and *Builder Bosses* tell Builders what to do and communicate with the Scanner Manager and the rest of the converter node. Each converter node has one Builder Boss and a team of several Builders.

A Builder is essentially an intelligent reception point, one that knows how to communicate with the Scanner CPUs and the Builder Boss. Scanner CPUs don’t know about converter nodes, but send their data to network links belonging to Builders, which have threads waiting to put data into the Builder buffers. The Scanner Manager tells the Scanner CPUs the network addresses (and port numbers if applicable) of the destinations.

### Builder

Instead of simple buffers, each converter node maintains a number of Builder objects which mimic event-builder machines with a capacity of one event each. A Builder also has one network link for each Scanner CPU, with the connection being established during creation, or, if applicable, a pre-defined permanent connection will be used. We don't plan to use connectionless network services. When a Scanner CPU transfers a fragment, the destination it specifies is a Builder’s network link. When data is detected coming in on a link, it is placed in a packet buffer which is then attached to the Builder.

**A Builder has:**

(1) *link,* a table of network links, each connected to a different Scanner CPU,
(2) *off,* a table of the buffer offsets (one per link),
(3) *log*, a network log,
(4) *copy,* a set of threads waiting for data from the links,
(5) *lock*, a semaphore controlling access to the Builder’s data,
(6) *scanners*, a bit mask showing which Scanners have sent data for the current event.

**A Builder can:**

(1) be created,
(2) be destroyed,

(3) start waiting for data from its links,
(4) stop waiting for data,
(5) print its connection log using a given file descriptor,
(6) return its scanner mask.

*boss* is supplied by the thread creating the Builder. Also supplied is a list of Scanner network IDs, used to create the entries of *link* and so form a network connection with each Scanner. The other variables and the *copy* threads are created at this time but not allowed to start execution. *scanners* is set to zero, the network log is empty.

Each thread *copy[i]* is an endless loop waiting for data using *link[i]:*

```
do while (1) {
  count = receive a transmission on link[i] into some free packet buffer;
  If (count > fragsize) {Report an error to boss;}
  take(lock);
    update log;
    set bit i in scanners;
    attach the buffer to the Builder;
  give(lock);
}.
```

Other threads must be locked out when the network log and scanner mask are updated because all of the Builder's threads share one instance of each. Also, the Builder can receive commands from the Builder Boss to dump the log, so that function has to use the lock as well. All the copy threads are always active once the Builder has been created, which makes it the responsibility of the Builder Boss and Scanner Manager to make sure that data is transmitted on the correct network links.

Destroying a Builder means killing the *copy* threads, closing all the links, and deallocating the other Builder variables.

## **Builder Boss**

This object maintains two sets of Builders. One set contains Builders which have finished receiving event data and which have been passed to the rest of the converter node (*unavailable*), and another set contains those Builders which are ready to receive new events and may already be doing so. (*available*). The Builder Boss manipulates the sets according to commands received from the Scanner Manager and the rest of the converter node.

**A Builder Boss has:**
(1) *available*, a set of Builders,
(2) *unavailable,* a set of Builders,
(3) *all*, a list of Builder pointers used to find the Builders no matter which set they're in,
(4) *smAddr,* the network address of the Scanner Manager,
(5) *log*, a log of messages sent to and received from the Scanner Manager,
(6) *globalID[],* the set of ID number for the Builders (assigned by the Scanner Manager).

**A Builder Boss can:**
(1) be created,
(2) be destroyed,
(3) move a Builder from one set to another,
(4) send, to a given file descriptor, a list of which Builders are in which sets,
(5) print its message log to a given file (C stdio FILE*).

A Builder Boss is created by a call to `init_receiver()` (see above), whose arguments give the locations of the memory areas to be used as packet buffers for event reception. A permanent network connection to the Scanner Manager is established (or used, if permanent). The Scanner Manager is informed of the number of Builders created and assigns a block of Builder ID numbers, sending them back to the converter node. These Builder ID numbers are just simple integers used in error messages and in communications between the Builder Boss and the Scanner Manager. The SM also sends the converter node the list of Scanner CPU addresses on the event transmission network. Each Builder is then created (see above) and placed in the *available* set. The *unavailable* set is empty.
A report is sent back to the Scanner Manager about any Builders which can't connect to all of the Scanners.

If all has gone well, the Builder Boss sends the Scanner Manger one message for each Builder on the *available* set, each message containing the Builder's ID number. Then the Builder Boss enters a loop awaiting commands from the Scanner Manager. When the SM sends an "event complete" message to the Boss, the corresponding Builder is removed from *available* and placed in the *unavailable* set, where it stays until the converter node is finished with it (see below).

Calls to `wait_next_event()` or `test_next_event()` test for a non-empty *unavailable* set. When they find one, the buffer index and scanner mask for the next Builder in that set are returned and the entry . When `release_event()` is called with the corresponding Builder number the Builder is removed from *unavailable,* added to *available,* and its ID is sent to the Scanner Manager. In addition the packet buffers are recycled. There is no permanent association between a Builder and a set of packet buffers.

A call to `kill_receiver()` causes all the Builders to be destroyed, then the Builder Boss itself.

**Partitions**

`init_receiver()` is called on a converter node when Run Control has selected that node to be part of a partition. However, the Scanner Manager does not tell the converter node which Scanner CPUs are in the partition until the Activate transition, at which time the Scanner Manager sends a bit mask to the Builder Bosses specifying the SCPUs. This means that normally each Builder forms a network link to every SCPU in the entire DAQ system.

**Threads**

Threads are trivially implemented as tasks under VxWorks. Under UNIX, an implementation of POSIX 1003.c threads (pthreads) is required. Solaris 2.5 and IRIX 6 have bundled pthreads packages. For earlier versions of those operating systems, I used Chris Provenzano's freeware implementation. See me to get a copy. At present Provenzano’s pthreads works under IRIX but not Solaris (the Solaris version has bugs in TCP/IP networking). Under Linux/GNU, the LinuxThreads package works well and is very close to standard pthreads. The only hitch here is that LinuxThreads uses the USR1 and USR2 signals, which conflicts with the converter node code. Luckily, the LinuxThreads package (now part of the standard Linux/GNU C library) contains instructions telling one how to recompile the package to use other signals.